J. Symbolic Computation (1998) **26**, 89–95
Article No. sy980202

An Algorithm for Solving the Factorization Problem in Permutation Groups

TORSTEN MINKWITZ

Deutsche Telekom AG, Bonn, Germany

The factorization problem in permutation groups is to represent an element g of some permutation group G as a word over a given set S of generators of G. For practical purposes, the word should be as short as possible, but must not be minimal. Like many other problems in computational group theory, the problem can be solved from a strong generating set (SGS) and a base of G. Different algorithms to compute an SGS and a base have been published. The classical algorithm is the Schreier–Sims method. However, for factorization an SGS is needed that has all its elements represented as words over S. The existing methods are not suitable, because they lead to an exponential growth of word lengths. This article presents a simple algorithm to solve the factorization problem. It is based on computing an SGS with elements represented by relatively short words over the generators.

1. Introduction

A popular game in the early 1980s was the Rubik's Cube. It is a cube with six faces, each coloured differently and which can be turned by hand in 90° steps. The mechanics of it requires each face to be subdivided into 3×3 little squares. With just a few random turns, one can get a mix of colours on each face. The goal of the game is to return the cube to its original state of unicoloured faces (see Hofstadter (1985) for a good description of the game).

For the purpose of this article, the interesting fact is that each turn of a face of the cube is indeed a permutation of the $6 \times 9 = 54$ little squares (only 48 are moved by the turns though, because the centre ones stay where they are). The six different 90° turns are the generators of a permutation group. To solve the game, one has to factorize the inverse of the group element corresponding to the given state of the cube. The algorithm presented in this article will solve the Rubik's Cube game. However, it is a general method for solving factorization problems in permutation groups. The Cube has nevertheless been an inspiration for this work.

Let G be a group generated by a set S of bijections of the finite set Ω, thus G is a permutation group. Then the subgroup $\mathrm{Stab}(G,x) := \{g \in G \mid x^g = x\}$ of G for $x \in \Omega$ is called the stabilizer of x and $Gx := \{x^g \mid g \in G\}$ is called the G-orbit of x. Let Ω be ordered in an arbitrary, but fixed way: $\Omega = \{x_1, \ldots, x_n\}$. The subgroup

$$G^{(i)} := \{g \in G | \forall_{1\le j<i} : x_j^g = x_j\} \tag{1.1}$$

of G is called the ith stabilizer (hence $G = G^{(1)}$ and $\mathrm{Stab}(G, x_1) = G^{(2)}$). The sequence

0747–7171/98/070089 + 07 $30.00/0

of all $G^{(i)}$, $1 \leq i \leq n$ is called a stabilizer chain. A strong generating set (SGS) is a subset R of G, such that for all $1 \leq i < n$ the equation $\langle R \cap G^{(i)} \rangle = G^{(i)}$ holds. A subset B of Ω is called a base for G, iff every $g \in G$ such that $x^g = x$ for all $x \in B$ is the identity. Obviously, Ω is a base. A more complete discussion of these definitions can be found in Butler (1991).

Let $B = \{b_1, \ldots, b_k\} \subset \Omega$ be a fixed and ordered base of G and Ω be ordered in such a way that $b_i = x_i, 1 \leq i \leq k$. Then partial maps ν_i, such that

$$\nu_i : \Omega \to G^{(i)}, \quad \text{and} \quad \begin{cases} \omega^{\nu_i(\omega)} = b_i & \text{if } \omega \in G^{(i)} b_i \\ \nu_i(\omega) & \text{undefined else} \end{cases} \tag{1.2}$$

hold, are tables of representatives of the right cosets of $G^{(i+1)}$ in $G^{(i)}$. This means that the defined entries of the tables (and therefore the cosets) are indexed by the elements of the $G^{(i)}$-orbit of b_i.

Given some $g \in G^{(1)}$ and a table ν_1, one can easily find an element of $G^{(2)}$. If g maps b_1 to ω_1, then $g\nu_1(\omega_1)$ is in $G^{(2)}$. Now, to compute an element of $G^{(3)}$, the same procedure is followed: if $g\nu_1(\omega_1)$ maps b_2 to ω_2, then $g\nu_1(\omega_1)\nu_2(\omega_2)$ is in $G^{(3)}$. Iterating this, one eventually reaches $G^{(k+1)} = \langle 1_G \rangle$ and

$$g\nu_1(\omega_1)\nu_2(\omega_2)\cdots\nu_k(\omega_k) = 1_G. \tag{1.3}$$

This way, given an ordered base B and all the tables ν_i, any $g \in G$ has a unique factorization

$$g = \nu_k(\omega_k)^{-1}\nu_{k-1}(\omega_{k-1})^{-1}\cdots\nu_1(\omega_1)^{-1}. \tag{1.4}$$

Therefore, if the images of the ν_i were all represented as words over the generators S, the factorization problem would be solved. One simply inverts the relevant words from the tables and concatenates the results.

The set $R := \bigcup_{1 \leq i \leq k} \text{Image}\,(\nu_i)$ is always an SGS for G, and the order of G can be computed through the product of the table sizes. This property serves as a termination criterion for the algorithm:

$$\texttt{TableFull}(G,\{\nu_i\}_i) := \left(|G| = \prod_{1 \leq i \leq k} |\text{Image}(\nu_i)| \right). \tag{1.5}$$

What is missing is an algorithm to compute the tables ν_i with all entries represented as words over the given generators S of a permutation group G.

2. The Algorithm

The task is to determine all entries of the tables as words over the generators S of G. For that, the group elements used during the algorithm will be represented as permutations and as words over S. However, the only two operations needed are inversion and multiplication. Therefore, this is not difficult. The basic idea of the algorithm is quite simple: use a lot of group elements to check the tables defining equation (1.2). If a needed table entry is not known yet, use the group element that caused the failure to find one. In detail, for some $i \in \{1 \ldots k\}$ and $t \in G^{(i)}$:

```
If ν_i(b_i^t) is defined
  Then
    r := tν_i(b_i^t);
```

```
  Else
    set ν_i(b_i^t) to t^{-1};
    r := 1_G;
Fi;
```

A post-condition of this `If`-clause is $r \in G^{(i+1)}$. Therefore, one can set t to r, increase i by 1 and apply the clause again. This is repeated until $i = |B|$. This simple procedure is now started over and over with a sequence of elements $t \in G^{(1)} = G$ which have short words. During the course the tables are filled. A simple way is to first use all elements with word length 1 over S, then all with word-length 2 and so on. Providing one new t is called a round. The procedure `Next()` returns the t for the next round. The parameters of `Next()` are S and the count of the round. One can stop whenever all tables are full (see equation (1.5)) and a predefined number of rounds n is reached.

Several obvious improvements apply:

1. If t has a shorter word than the current $\nu_i(b_i^t)$, then $\nu_i(b_i^t)$ can be set to t^{-1}. Inversion keeps word lengths invariant. (In practice, it is even a good idea to set when the two word lengths are equal, because the resulting jitter keeps the ν_i from getting "stuck" in a bad state.)
2. If t^{-1} is the shortest known candidate for $\nu_i(\omega_i)$, then t is often a good candidate for $\nu_i(b_i^{t^{-1}})$.
3. The computation for each round is accelerated significantly, if a new round is started whenever t has a word length exceeding some limit l or $t = 1_G$.

The `If`-clause is thus modified to become a recursive procedure. A tilde $\sim$ is used to indicate parameters that are passed by reference and can thus be modified.

```
Step := Procedure(G, B, i, t, ∼r, ∼{ν_j}_j)
  If ν_i(b_i^t) is defined
    Then
      r := tν_i(b_i^t);
      If Wordlength(t) < Wordlength(ν_i(b_i^t))
        Then
          set ν_i(b_i^t) to t^{-1};
          Step(G, B, i, t^{-1}, ∼r', ∼{ν_j}_j);
      Fi;
    Else
      set ν_i(b_i^t) to t^{-1};
      Step(G, B, i, t^{-1}, ∼r', ∼{ν_j}_j);
      r := 1_G;
  Fi;
```

A round starting at level i is specified through:

```
Round := Procedure(G, B, l, c, ∼{ν_j}_j, ∼t)
  i := c;
  While (t ≠ 1_G) and (Wordlength(t) < l) Do
    Step(G, B, i, t, ∼r, ∼{ν_j}_j);
```

```
    t := r;
    i := i + 1;
  Od;
```

The algorithm resulting from these improvements already performs quite well:

```
SGSWord := Procedure(G, S, B, l, n, ∼{ν_i}_i)
  set all ν_i undefined everywhere, except for ν_i(b_i) = 1_G;
  count := 0;
  While (count < n) or (not TableFull(G, {ν_i}_i)) Do
    t := Next(S, count);
    count := count + 1;
    Round(G, B, l, 1, ∼{ν_i}_i, ∼t);
  Od;
```

However, there are still a number of deficiencies. A very significant one is the waste caused by not making full use of having found a new and good (meaning short word) image of ν_i. This can be compensated for by stopping every s rounds and then looking at the new entries in ν_i for each i. These are multiplied with other entries and the results are new t to use for a `Round()` at level i. This will often improve the images of the ν_i. A good choice for s is difficult, but the square of the size of the base $|B|^2$ is usually not too bad.

There are cases, in which the algorithm will still perform rather badly. A prominent example is the symmetric group S_N, when generated as

$$S_N := \langle (1,2), (2,3), (3,4), \ldots, (N-2, N-1), (N-1, N) \rangle. \tag{2.1}$$

Since there are so many generators, it takes a long time to reach a round with a t of sufficient word length. For each G and S, one needs a particular word length for the t to cover all cosets at all levels. Also, `SGSWord()` in itself is a bad finisher. It may take a long time to find the last few entries. The solution to these problems is to also run another procedure every s rounds, which specializes in filling all tables (see procedure `FillOrbits()` below). It is a very simple algorithm, which looks at all entries of all tables and for each level i tries to find entries for hitherto undefined $\nu_i(\omega)$.

The parameter l should be set small in the beginning. It can be increased slowly, while the tables are still not full. A small value for l reduces the run-time for each round, because the rounds are terminated earlier. The complete algorithm is specified by the following pseudocode procedures:

```
SGSWordQuick := Procedure(G, S, B, n, s, ∼l, ∼{ν_i}_i)
  set all ν_i undefined everywhere, except for ν_i(b_i) = 1_G;
  count := 0;
  While (count < n) or (not TableFull(G, {ν_i}_i)) Do
    t := Next(S, count);
    count := count + 1;
    Round(G, B, l, 1, ∼{ν_i}_i, ∼t);
    If (count mod s) = 0
      Then
        Improve(G, B, l, ∼{ν_i}_i);
```

```
        If (not TableFull(G, {ν_i}_i))
          Then
            FillOrbits(G, B, l, ~{ν_i}_i);
            l := 5/4 l;
        Fi;
    Fi;
  Od;

Improve := Procedure(G, B, l, ~{ν_i}_i)
  For j From 1 To |B| Do
    For x In Image(ν_j) Do
      For y In Image(ν_j) Do
        If (x or y are new in Image(ν_j))
          then
            t := x y;
            Round(G, B, l, j, ~ν_i, ~t);
        Fi;
      Od;
    Od;
  Od;

FillOrbits := Procedure(G, B, l, ~{ν_i}_i)
  For i From 1 To |B| Do
    O := {b_i^y : y ∈ Image(ν_i)}; (partial orbit already found)
    For x In ∪_{i<j≤|B|} Image(ν_j) Do
      For p In O^x − O Do (walk through new points of the orbit)
        t := ν_i(p^{x^{-1}})x;
        If WordLength(t) < l
          Then
            set ν_i(p) to t^{-1};
        Fi;
      Od;
    Od;
  Od;
```

The quality of the resulting tables is measured in maximum word lengths for a factorization. It can be found through:

$$\sum_{1 \le i \le k} \text{Max } (\{Wordlength(y) : y \in Image(\nu_i)\}). \tag{2.2}$$

The smaller this value the better. To improve it, one can set the parameter n to a higher value or rerun the algorithm using a different base or base ordering, both of which matter a great deal here.

Table 1. Run-times for `SGSWordQuick`.

Group	Degree	Order	$\lvert B\rvert$	n	Time (s)	Maximum word length
$PGL_3(8)$	73	1.6×10^7	4	10^4	88	48
				3×10^4	285	46
Rubik's Cube	48	4.3×10^{19}	18	10^4	110	165
				3×10^4	276	155
				10^6	7289	144
CubeGray_5	32	2.1×10^{26}	28	10^4	124	415
				3×10^4	312	343
				10^6	6643	249
CubeGray_6	64	3.4×10^{70}	60	3×10^4	475	1988
				10^6	8965	936
CubeGray_7	128	8.1×10^{177}	124	3×10^5	12807	4893
				10^6	47114	3843
S^*_{20}	20	2.4×10^{18}	19	10^4	116	403
S^{**}_{20}	20	2.4×10^{18}	19	10^3	11	37
S^*_{50}	50	3×10^{64}	49	10^5	6370	3449
S^{**}_{50}	50	3×10^{64}	49	10^4	551	97

3. Results and Conclusion

To test the algorithm, it was applied to a number of groups with different characteristics. The family of groups CubeGray_N[†] was used here, although these groups are not generally known. However, they were good to test the algorithm, because their bases are large, but the word lengths in Image (ν_i) can be made to remain almost stable for growing i. Schreier–Sims will not achieve this. Those groups contrast well with the symmetric groups. Here S^*_N denotes the symmetric group on n points with generators as in (2.1), whereas S^{**}_N denotes the same group generated by the permutations on $\Omega := \{1, \ldots, N\}$:

$$S^{**}_N := \langle (1,2), (1,3), \ldots, (1,N-1), (1,N) \rangle. \tag{3.1}$$

The symmetric groups are the extreme case to demonstrate the effect of using different generators. The group $PGL_3(8)$ demonstrates good behaviour in the case of a small base but with large degree.

All cpu-times in Table 1 were achieved on a Sun Sparc IPX with programs written in the GAP programming language. It has been demonstrated that the new algorithm performs well for a lot of different permutation groups. It could be included in group theory systems like MAGMA (see Cannon and Bosma (1994)) or GAP (see Schönert *et al.* (1992)) to improve their abilities further.

[†]The group CubeGray_N is a subgroup of the group of permutations of the vertices of the N-dimensional hypercube graph. All vertices can be labelled by a unique bit-string of length N, such that the strings of neighbouring vertices differ by only one bit. It follows that a Hamilton circle (a non-intersecting cyclic path that reaches all vertices) can always be found along a Gray-code of labels. This is of course also true for the two $(N-1)$-dimensional subcubes that are defined by fixing any of the N digits of the bit-strings. The group CubeGray_N is the group of permutations generated by shifts along the two Gray-code Hamilton circles of the vertices of the $(N-1)$-dimensional subcubes in each of the N dimensions.

Acknowledgements

I wish to thank my former advisor T. Beth and also S. Egner and A. Nückel, all working at the Institute of Algorithms and Cognitive Systems in Karlsruhe.

References

Atkinson, M.D. (Ed) (1984). *Computational Group Theory*. Academic Press.
Butler, G. (1991). *Fundamental Algorithms for Permutation Groups*. Springer-Verlag: LNCS 559.
Cannon, J., Bosma, W. (1994). *Handbook of Magma Functions*. University of Sidney.
Hofstadter, D. (1985). *Metamagical Themas: Questing for the Essence of Mind and Pattern*. Basic Books.
Leon, J.S. (1980). On an algorithm for finding a base and a strong generating set for a group given by generating permutations. *Math. of Computation*, **35**, 941–974.
Schönert, M. *et al.* (1992). *GAP: Groups, Algorithms and Programming*. Lehrstuhl D für Mathematik, RWTH Aachen.
Sims, C.C. (1970). Computational methods in the study of permutation groups. In Leech, J., ed., *Computational Problems in Abstract Algebra*, pp. 169–183. Pergamon.

Originally Received 7 June 1994
Accepted 2 March 1998